A Tour of Various TPL Options

Joseph E. Hoag

Parallel Computing Platform Group

Microsoft Corporation

Abstract: The Task Parallel Library (TPL), released as part of the new parallel programming support in .NET 4, provides various options to the user via the TaskCreationOptions, TaskContinuationOptions and ParallelOptions classes. These options, accompanied by examples of correct and incorrect uses of them, are discussed in this article.

Table of Contents

Introduction

The Task Parallel Library (TPL), released as part of the new parallel programming support in .NET 4, provides various options to the user via the TaskCreationOptions, TaskContinuationOptions and ParallelOptions classes. These options, accompanied by examples of correct and incorrect uses of them, are discussed in this article.

TaskCreationOptions

By default, a Task is created as follows:

- No "Fairness" in Task scheduling is assumed – Tasks may be scheduled in a Last-In-First-Out (LIFO) fashion in order to achieve better performance due to cache locality
- The Task does not have a logical parent
- The Task is assumed to be short-running

One can use the TaskCreationOptions when creating a Task to override that default Task behavior.

PreferFairness

The general intent of this option is to request that Tasks be scheduled fairly with regards to other Tasks also created with this option. This typically results in an ordering that approximates first-in-first out, or FIFO. It should be noted that **the response to this option is entirely at the discretion of the TaskScheduler used to process the Task** – some TaskScheduler implementations may completely ignore the PreferFairness option. However, the default TaskScheduler does recognize and respond to the PreferFairness option, and we will discuss some specifics of that handling herein.

Let us first review some of the inner workings of the CLR ThreadPool, upon which the default TaskScheduler is based. The ThreadPool contains a local Task queue for each worker thread, as well as a single global Task queue (see Figure 2 for a graphical representation of the ThreadPool). When a worker thread completes the execution of one Task, it will attempt to execute another Task from its local queue, **in LIFO order**. When a worker thread's local queue "runs dry", the thread will look for work elsewhere, first from the global Task queue (**in FIFO order**), then from the Task queues of other worker threads (**in FIFO order**). When one worker thread gets its work from another worker thread's Task queue, this is known as *work-stealing*.

(For a more in-depth discussion of work-stealing and TPL's use of the ThreadPool, see articles [1], [2] and [3].)

When a Task is queued to the default TaskScheduler from an "external" context (i.e., not from a ThreadPool thread's context), that Task is always queued to the ThreadPool's global Task queue. However, when a Task is queued from the context of ThreadPool thread, one of two things can happen:

- If the Task was created with the PreferFairness option, it will be queued to the global Task queue.

- Otherwise, the Task is queued to the local queue of the worker thread under whose context the queuing was requested.

In effect, the presence of the PreferFairness option guarantees that a Task will be queued to the ThreadPool's global Task queue. And since the ThreadPool's global Task queue is processed in roughly FIFO order, Tasks created with the PreferFairness option will be de-queued in roughly the same order as that in which they were queued.

Consider the following code:

```
public static void Main(string[] args)
{
    Task A = Task.Factory.StartNew( () =>
    {
        Task B = Task.Factory.StartNew( () => {…});
        Task C = Task.Factory.StartNew( () => {…}, TaskCreationOptions.PreferFairness);
    });
    …
}
```

In the example above:

- Task A is queued to the ThreadPool's global Task queue, since it is created from an external context. It is soon picked up for execution by a worker thread.
- Task B is queued to the local Task queue of the worker thread running Task A, since it is created from the context of Task A's ThreadPool thread.
- Task C, even though it is created from the context of Task A's ThreadPool thread, is still queued to the ThreadPool's global Task queue, because it is created with the PreferFairness option.

As previously mentioned, there are reasons why this option is called "PreferFairness" instead of "GuaranteeFairness":

- While the default TaskScheduler recognizes this option, custom TaskSchedulers may choose to completely ignore it.
- Even if the default TaskScheduler is being targeted, there is a certain amount of unpredictability inherent in any parallel system. While the use of PreferFairness can guarantee the order in which Tasks are de-queued from the scheduler's queue in .NET 4, it cannot guarantee the order in which the Tasks are actually executed. Task X may be de-queued before Task Y, but due to various nondeterministic activities in the system (interrupts, context switches, etc…) that does not necessarily mean that Task X will begin execution before Task Y.
- While the ThreadPool's global Task queue is processed in precise FIFO order in .NET 4, there is no guarantee that this will always be the case in future releases. Future versions of the ThreadPool may process the global Task queue in *approximate* FIFO order instead of *precise* FIFO order. Thus the frequent use of the phrase "roughly FIFO order" to describe this ordering.

For more information on the PreferFairness option, see Stephen Toub's blog post on that subject [4].

Useful Application: Ensuring Progress

One good use of PreferFairness is a situation where a Task's progress depends upon the execution of sub-Tasks, but the Task is somehow blocked from explicitly waiting on the sub-Tasks. (A Wait or WaitAll call on the sub-Tasks would result in them being inlined[1].) The Task can't make progress until its sub-Tasks execute. Here is an example of this situation involving speculative execution:

```
Task<int> outer = Task<int>.Factory.StartNew(() =>
{
   CancellationTokenSource cts = new CancellationTokenSource();
   CancellationToken token = cts.Token;

   Task<int>[] tasks = new Task<int>[3];
   tasks[0] = Task.Factory.StartNew(() => {methodA(token);}, token);
   tasks[1] = Task.Factory.StartNew(() => {methodB(token);}, token);
   tasks[2] = Task.Factory.StartNew(() => {methodC(token);}, token);

   int winnerIndex = Task.WaitAny(tasks);
   cts.Cancel();
   return tasks[winnerIndex].Result;
});
```

In the example above, Task **outer** launches three sub-Tasks, and waits to see which will be first to return an answer. The winner's answer is recorded, and any remaining sub-Tasks are canceled. (It is assumed that methodA/B/C all internally pay attention to **token**, and will quit once it is signaled.) It is important to note that Task.WaitAny() does not perform inlining of its target Tasks; if it did, it might be busy inlining one while another completed, and would thus be tardy in recognizing the "winner". Thus Task **outer** cannot make progress until its sub-Tasks execute, and all of its sub-Tasks are queued to **outer**'s local work-stealing queue.

If the ThreadPool is lightly loaded, as in Figure 1, the code above will work well. The idle worker threads will quickly "steal" **tasks[0]**, **tasks[1]** and **tasks[2]**, and the application will make progress. Though there will be some overhead involved with the work-stealing, which could have been avoided through the judicious use of the PreferFairness option, it will be minimal.

However, if the ThreadPool is heavily loaded, as in Figure 2, the code above may not work very well – all of the other worker threads are busy. And if the ThreadPool injects additional worker threads, they will pick up their work from the global Task queue, not Worker Thread 1's local Task queue. Basically, the **outer** Task *will make no progress until all until the global Task queue is exhausted*, and in a busy system

[1] "Inlining" a Task, in this context, means executing a queued Task immediately, rather than waiting for it to be de-queued and executed by the applicable TaskScheduler. Inlining is an important optimization mechanism to allow Task.Wait and Task.WaitAll to perform useful work while waiting.

it might be some time before the global queue is exhausted. In fact, if the incoming rate of work exceeds the processing rate of the work items, the **outer** Task will never complete.

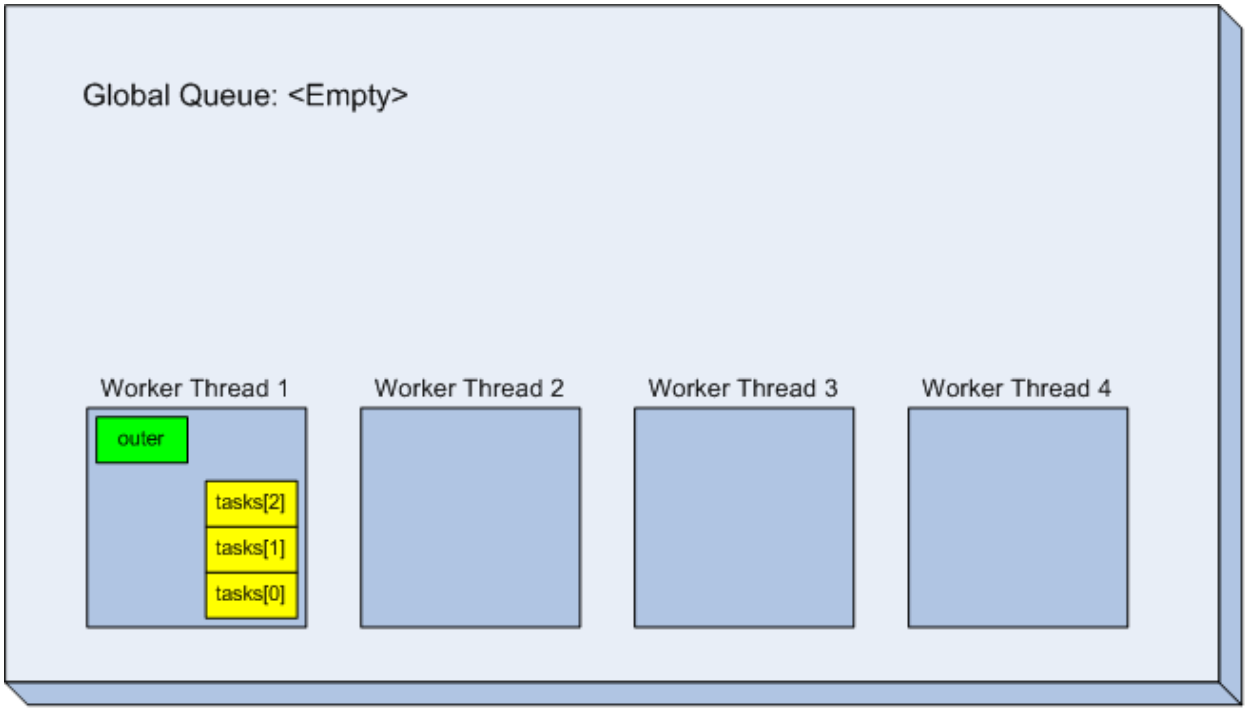


Figure 1: Sample Code Run on Lightly Loaded ThreadPool (green = running, yellow = queued)

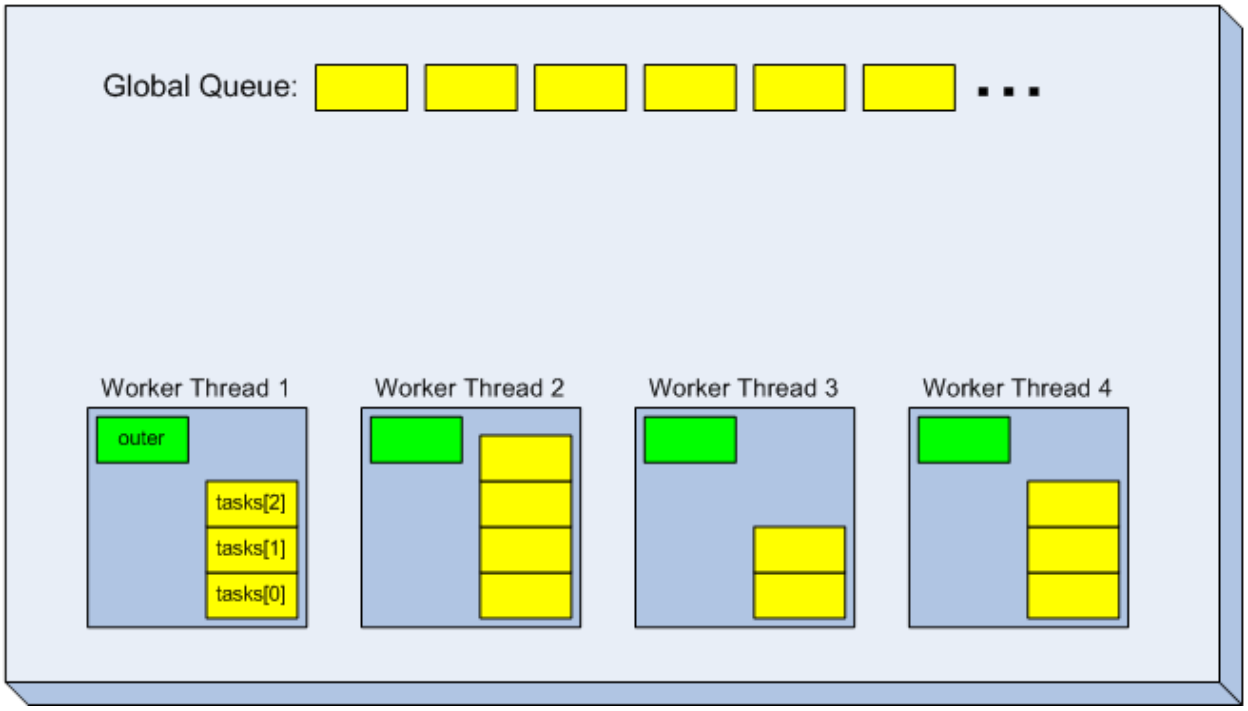


Figure 2: Sample Code Running on Heavily Loaded ThreadPool (green = running, yellow = queued)

If you are worried about your code making progress in a busy system, it would be better to forego the local Task queue for the sub-Tasks, and use the PreferFairness option to queue them directly onto the global work queue (additions highlighted[2]):

```
Task<int> outer = Task<int>.Factory.StartNew(() =>
{
   CancellationTokenSource cts = new CancellationTokenSource();
   CancellationToken token = cts.Token;

   Task<int>[] tasks = new Task<int>[3];
   tasks[0] = Task.Factory.StartNew(() => {methodA(token);},
      token, TaskCreationOptions.PreferFairness, TaskScheduler.Default);
   tasks[1] = Task.Factory.StartNew(() => {methodB(token);},
      token, TaskCreationOptions.PreferFairness, TaskScheduler.Default);
   tasks[2] = Task.Factory.StartNew(() => {methodC(token);},
      token, TaskCreationOptions.PreferFairness, TaskScheduler.Default);

   int winnerIndex = Task.WaitAny(tasks);
   cts.Cancel();
   return tasks[winnerIndex].Result;
});
```

In the code above (as graphically illustrated in Figure 3), the sub-Tasks are all queued to the global work queue, which is processed in roughly FIFO order by newly spun-up worker threads and worker threads that "run dry". Therefore, *some* progress is guaranteed even if the system is very busy.

Just as an aside, the modified code above could be simplified by taking advantage of the customizable TaskFactory feature in .NET 4. The creation of the inner Tasks could be accomplished with this code:

```
var tf = new TaskFactory(token, TaskCreationOptions.PreferFairness,
                         TaskContinuationOptions.None, TaskScheduler.Default);
tasks[0] = tf.StartNew( () => {methodA(token);});
tasks[1] = tf.StartNew( () => {methodB(token);});
tasks[2] = tf.StartNew( () => {methodC(token);});
```

[2] There is no overload of StartNew() that accepts CancellationToken and TaskCreationOptions parameters, but not a TaskScheduler parameter. Thus, we needed to provide a TaskScheduler parameter along with our TaskCreationOptions parameter.

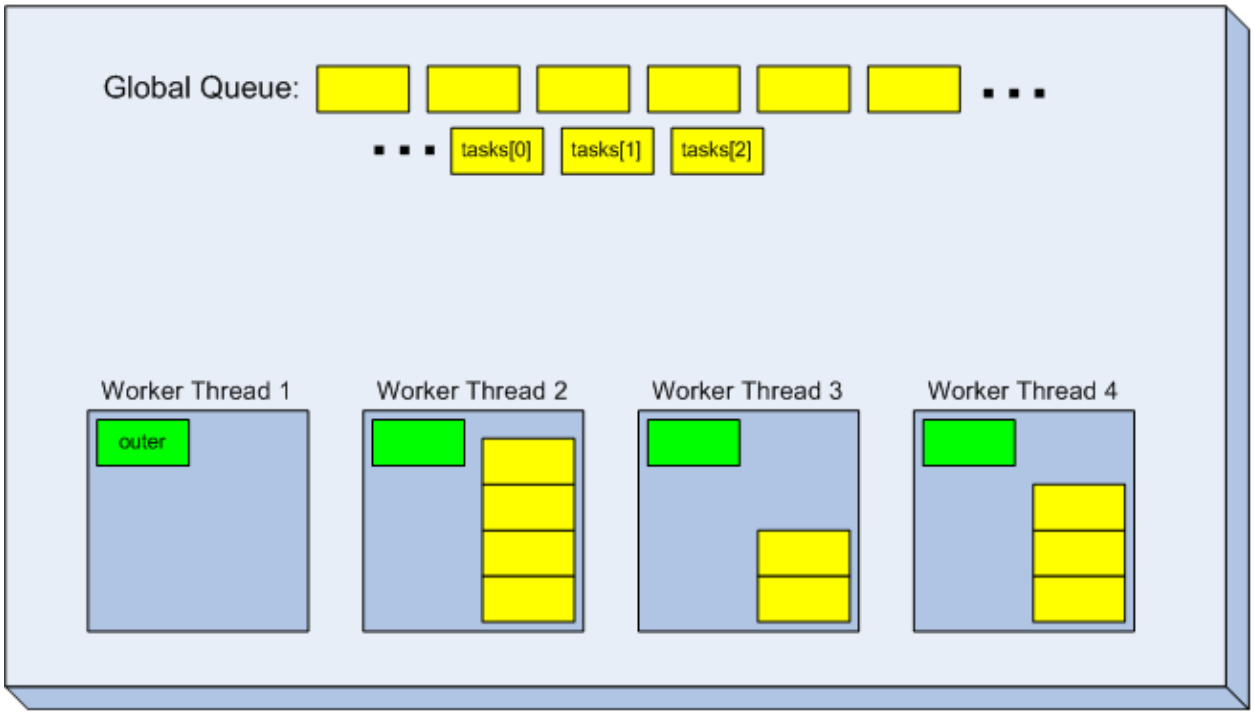


Figure 3: Sample Code Using PreferFairness (green = running, yellow = queued)

LongRunning

By default, Tasks are assumed to be brief operations. If Tasks are queued to the TaskScheduler that perform a lot of blocking (allowing for the possibility of significant wait periods) or that will in general run for long periods of time (potentially preventing the various work queues from being processed), the LongRunning value can be supplied at Task creation time as a hint to the TaskScheduler that it may want to increase the concurrency level to allow additional Tasks to execute. This is similar in concept to WT_EXECUTELONGFUNCTION in the Win32 thread pool and CallbackMayRunLong in the newer Vista/Windows Server 2008 thread pool. This is merely a hint to the target TaskScheduler, which may choose to **ignore it entirely**, supply a hint to the thread injection/retirement mechanism, implement it with a dedicated thread not otherwise part of the pool, implement it with a custom pool, or any other implementation deemed appropriate.

In the current implementation of .NET 4, when the default TaskScheduler is instructed to queue a Task created with the LongRunning option, it will spin up a standalone thread to run the Task, rather than submitting the Task to the ThreadPool. The effect can be seen when running the following (admittedly contrived) code:

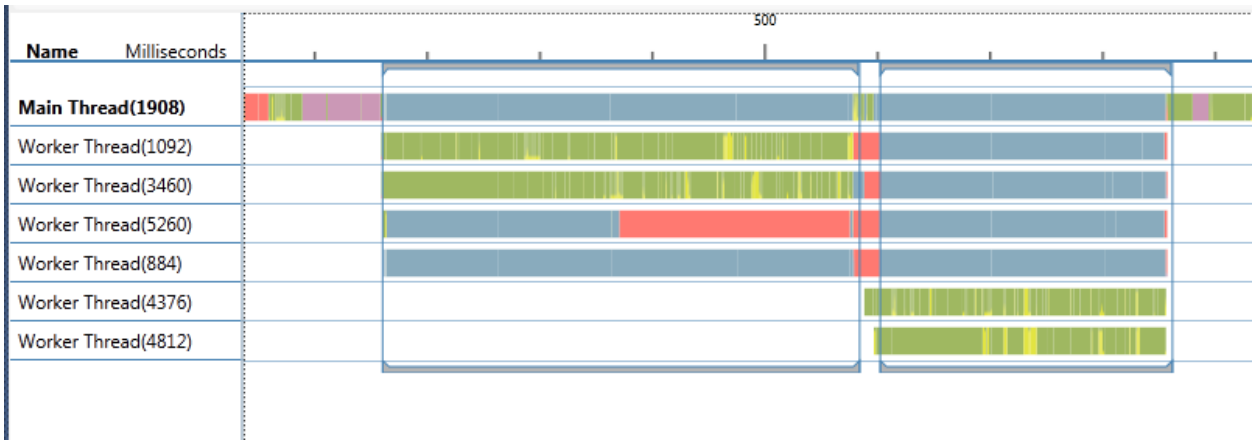
```
static void LongRunningTest(bool longRunningOn)
{
   bool stopLongRunningTasks = false;
   TaskCreationOptions options =
      longRunningOn ?
            TaskCreationOptions.LongRunning :
            TaskCreationOptions.None;

    Task t1 = Task.Factory.StartNew(() => { while (!stopLongRunningTasks) ;},
options);
    Task t2 = Task.Factory.StartNew(() => { while (!stopLongRunningTasks) ;},
options);

   Parallel.For(0, 10, _ => { Thread.Sleep(100); });

   stopLongRunningTasks = true; // kill t1 & t2
   Task.WaitAll(t1, t2);
}
```

If the **longRunningOn** parameter is true, then the LongRunning option is used to create **t1** and **t2**. Otherwise, **t1** and **t2** are created in default mode. This method was run twice (first with **longRunningOn** = false, next with **longRunningOn** = true) and the following profile information was gathered:



(The top thread is the main thread, the next four are ThreadPool worker threads, and the bottom two are "standalone" threads spun up when Tasks **t1** and **t2** are created with the LongRunning option.)

In the first run (without the LongRunning option), it can be seen that two of the four ThreadPool threads are occupied running **t1** and **t2**. In the second run (with the LongRunning option), **t1** and **t2** were run on threads outside of the ThreadPool, and all ThreadPool threads participated in the Parallel.For loop. The Parallel.For loop was able to complete more quickly in the second run.

Useful Application: Special Treatment for Any Long-Running Task

The LongRunning option is useful for any long-running Task. As a general rule of thumb, any Task that will run for more than a few seconds should be created with the LongRunning option. Some particularly applicable scenarios for which you should consider using the LongRunning option include the following:

- Tasks that perform CPU-intensive operations for longer than a few seconds.
- Tasks that spend a lot of time waiting on IO operations.
- Tasks that spend a lot of time sleeping – for example, "reporter" Tasks that wake up every so often and publish some state.
- Tasks that wait on each other – for example, producer/consumer scenarios in which producer Tasks and consumer Tasks use a BlockingCollection to communicate.

As always, measurement is important – don't just assume that using the LongRunning option will improve the performance of your application! When considering the LongRunning option, take measurements with and without its use, and compare them to determine the actual effect of the LongRunning option.

AttachedToParent

By default, a Task is parentless. By specifying this option when creating a Task, you establish a parent-child relationship between the currently executing Task and the newly created Task. For example:

```
Task t1 = Task.Factory.StartNew(delegate
{
        …
        Task t2 = Task.Factory.StartNew(delegate {…},
                TaskCreationOptions.AttachedToParent);
        Task t3 = Task.Factory.StartNew(delegate {…});
});
```

In the code snippet above, **t2** is **t1**'s child and **t1** is **t2**'s parent. Tasks **t1** and **t3** have no parent-child relationship with each other.

This ability to establish a parent-child relationship between Tasks is a feature that brings with it this added behavior:

- A parent will not complete until all of its children complete.
- Unless a parent observes a child's exception directly, through Wait(), that child's exception will be propagated to its parent.

The hierarchical structure afforded by establishing parent-child relationships between Tasks leads us to refer to this technique as *structured parallelism*.

Let's examine another code sample:

```
Task outer = Task.Factory.StartNew(delegate
{
        Task inner1 = Task.Factory.StartNew( delegate {throw new Exception("boo!");},
                TaskCreationOptions.AttachedToParent);
        Task inner2 = Task.Factory.StartNew( delegate {throw new Exception("gotcha!");},
                TaskCreationOptions.AttachedToParent);
        Task inner3 = Task.Factory.StartNew( delegate {throw new Exception("ouch!");});
        Task inner4 = Task.Factory.StartNew( delegate {throw new Exception("tricky!");},
                TaskCreationOptions.AttachedToParent);

        // Wait for inner4 to complete, w/o calling Wait()
        ( (IAsyncResult) inner4).AsyncWaitHandle.WaitOne();
        Console.WriteLine("inner4's exception = {0}", inner4.Exception);

        try
        {
                inner1.Wait();
        }
        catch(Exception e)
        {
                Console.WriteLine("inner1 threw {0}", e);
        }
});

try
{
        outer.Wait();
}
catch(Exception e)
{
        Console.WriteLine("outer threw {0}", e);
}
```

In the code above, what happens to the exceptions thrown by Tasks **inner1**, **inner2**, **inner3** and **inner4**?

- **inner1** is a child Task whose exception ("boo!") is observed by its parent when **outer** calls **inner1.Wait()**. Therefore the "boo!" exception will propagate no further.
- **inner2** is a child Task whose exception ("gotcha!") is *not* observed by its parent (**outer**), therefore the exception is propagated to its parent. The **outer.Wait()** call will throw an AggregateException that wraps the "gotcha!" exception.
- **inner3** is a "detached" (non-child) Task whose exception ("ouch!") is not observed. **inner3**'s exception will not be propagated to **outer**, because **outer** is not **inner3**'s parent. Rather, as is the case with any unobserved Task exception, the "ouch!" exception will be "bubbled up" to the finalizer thread and thrown from there.

- The handling of **inner4's** exception is the most complicated to follow. Since **inner4**'s exception was "observed" via **outer**'s call to **inner4.Exception**, this exception is considered "handled" and will not precipitate an "unobserved Task exception" exception being thrown from the finalizer. However, the reading of **inner4.Exception** does not count as "parent observation" – only a Wait() call can accomplish that. So, like **inner2**'s exception, **inner4**'s exception will be propagated to its parent, and will be included in the AggregateException thrown from the **outer.Wait()** call.

One more piece of sample code:

```
Task outer = Task.Factory.StartNew( () =>
{
        Task zombie = new Task( () => {}, TaskCreationOptions.AttachedToParent);

        Task inner = Task.Factory.StartNew( () =>
        {
                while(true) ;
        }, TaskCreationOptions.AttachedToParent);

        Thread.Sleep(1000);
});

outer.Wait();
```

In the code above, how long does it take for the **outer.Wait()** call to complete? No time at all? One second? In fact, the answer is that **outer** will never complete, because it has not one but **two** child Tasks that will never complete:

- Task **inner** will never complete because its delegate consists of an infinite loop.
- Perhaps less intuitively, Task **zombie** will never complete *because it never gets queued*. It gets constructed, but never handed off to a TaskScheduler. And the AttachedToParent behavior is established at construction time, not at queuing time.

A parent Task cannot complete until all of its child Tasks have completed. So the **outer.Wait()** call will hang forever (or at least until you Ctrl-C the program).

Useful Application: Tree Traversal

Since a tree is a hierarchically organized data structure, it is natural and efficient to traverse a tree using structured parallelism:

```
static void ProcessTree<T>(Node<T> root)
{
	Task rootTask = new Task ( () => {ProcessTreeNode(root);});
	rootTask.RunSynchronously();
	rootTask.Wait();
}

static void ProcessTreeNode<T>(Node<T> node)
{
	if(node == null) return;

	// don't waste time creating/running a Task if child is null
	if(node.Left != null)
	{
		Task.Factory.StartNew( () =>
		{
			ProcessTreeNode(node.Left);
		}, TaskCreationOptions.AttachedToParent);
	}

	// don't waste time creating/running a Task if child is null
	if(node.Right != null)
	{
		Task.Factory.StartNew( () =>
		{
			ProcessTreeNode(node.Right);
		}, TaskCreationOptions.AttachedToParent);
	}

	Process(node);
}
```

In the **ProcessTree** method above, the **rootTask** will not complete until all of its children (and grandchildren, and great-grandchildren, etc…) complete. And the **rootTask.Wait()** call will throw any exception generated from any descendent Task. You *could* write the **ProcessTree** method without using structured parallelism, but it would involve more work and look less "natural" (necessary additions highlighted):

```
static void ProcessTreeNode<T>(Node<T> node)
{
	if(node == null) return;
	List<Task> taskList = new List<Task>();

	// don't waste time creating/running a Task if child is null
```

```
        if(node.Left != null)
        {
            taskList.Add(
                Task.Factory.StartNew( () =>
                {
                    ProcessTreeNode(node.Left);
                })
            );
        }

        // don't waste time creating/running a Task if child is null
        if(node.Right != null)
        {
            taskList.Add(
                Task.Factory.StartNew( () =>
                {
                    ProcessTreeNode(node.Right);
                })
            );
        }

        Process(node);

        if(!taskList.IsEmpty) Task.WaitAll(taskList.ToArray());
    }
```

Anti-Pattern: Don't Create Library Tasks with the AttachedToParent Option

If you are writing a library, your code should not create AttachedToParent Tasks (at least not from the top level). Consider what might happen if your library method looked like this:

```
    static int reportingTaskStarted = 0;
    static int someLibraryMethod(int X, int Y)
    {
        // Start a reporting Task the first time that we're called
        if( (reportingTaskStarted == 0) &&
           (Interlocked.CompareExchange(ref reportingTaskStarted, 1, 0) == 0))
        {
            Task.Factory.StartNew( () => {someLongReportingLoop();},
                TaskCreationOptions.AttachedToParent);
        }

        int result = … some sort of calculation involving X and Y …;
        return result;
    }
```

Now imagine that some code calling into your library does this:

```
Task<int> A = Task.Factory.StartNew( () =>
{
        int X = …;
        int Y = …;
        return someLibraryMethod(X, Y);
});
```

The end result is that someone reading **A.Result** (which blocks waiting for Task **A** to complete) after the initial call to **someLibraryMethod**() will have to wait for their answer until the reporting Task spawned in **someLibraryMethod**() completes, which may be never. Or, perhaps even worse, the **someLongReportingLoop**() method could throw an exception, which would be propagated up to Task **A**.

None

This option signifies to the system that default Task behavior is desired:

- The Task is not a child Task (no AttachedToParent option)
- The Task is not long-running (no LongRunning option)
- There is no desire for "fairness" in Task execution order

Useful Application: Requesting Default Options When a TaskCreationOptions Parameter is Required

The TaskCreationOptions.None option is useful in situations where a TaskCreationOptions parameter is required, but no deviation from the default is desired. For example, if you want to specify a CancellationToken and a TaskScheduler when starting a Task, you must also provide a TaskCreationOptions parameter:

```
Task t1 = Task.Factory.StartNew(delegate {…},
        someCancellationToken,
        TaskCreationOptions.None,
        someTaskScheduler);
```

In the example above, the use of TaskCreationOptions.None signifies "I'm fine with the default Task behavior".

TaskContinuationOptions

This set of options is used when creating continuation Tasks. These options can be summarized by breaking them into 4 categories:

- Task Creation Options: the same options (PreferFairness, LongRunning, AttachedToParent) as described under TaskCreationOptions above.
- Triggering Options: these allow the user to govern whether a continuation will run or be canceled, based on the completion status of its antecedent.
- ExecuteSynchronously: controls whether or not the continuation is run "inline".
- None: shorthand for "default options"

By default, continuation Tasks have no creation options, no triggering options (which means they *always* trigger regardless of the final state of their predecessor), and are not inlined. A user can specify one or more TaskContinuationOptions to alter that default behavior.

Task Creation Options

The TaskContinuationOptions associated with continuation Task behavior – PreferFairness, LongRunning and AttachedToParent – are identical in function to their counterparts already described in the TaskCreationOptions. These apply to the behavior of the newly created continuation Task.

Triggering Options

A Task will complete in one of three TaskStatus values: Faulted, Canceled or RanToCompletion. By default, a continuation Task will always fire upon completion of its antecedent, regardless of the final status of the antecedent.

The triggering options allow a user to customize the conditions under which a continuation Task will fire. When triggering options are specified at the creation of a continuation Task, the continuation Task will only fire if those triggering options match the final status of the continuation Task's antecedent. If there is a mismatch between the antecedent's completion status and the continuation Task's triggering options, then the continuation Task will be canceled.

It should be noted that TaskFactory.ContinueWhenAll and TaskFactory.ContinueWhenAny, which accept multiple antecedents, do not allow the use of triggering options when creating continuations. An ArgumentOutOfRange exception will be thrown if this is attempted. Currently, only the ContinueWith methods, which accept a single antecedent, allow the use of continuation triggering options. Support for the use of triggering options when creating continuation Tasks with ContinueWhenAll and ContinueWhenAny may be added in a future .NET release.

This subsection describes the triggering options supported in TaskContinuationOptions.

NotOnRanToCompletion

This option instructs the continuation Task to **not** fire if the antecedent runs to completion. If the antecedent runs to completion, the continuation Task will be canceled. Consider the following:

```
Task antecedent = Task.Factory.StartNew( () => {});
bool ran = false;
Task c1 = antecedent.ContinueWith( _ => {ran = true;},
        TaskContinuationOptions.NotOnRanToCompletion);
```

Since **antecedent** runs to completion, and **c1** is created with the **NotOnRanToCompletion** triggering option, **c1** is canceled rather than run. The final value of **ran** is **false**.

Useful Application: Mopping up after an antecedent fails to complete.

This option can be used to run a "mop-up" continuation if the antecedent fails to complete.

```
Task t1 = Task.Factory.StartNew( () => { … });

Task c1 = t1.ContinueWith(antecedent =>
{
        PublishProblemReport(…);
},
TaskContinuationOptions.NotOnRanToCompletion);
```

NotOnFaulted

This option instructs the continuation Task to **not** fire if the antecedent faults (i.e., if the antecedent's delegate throws an exception). If the antecedent faults, the continuation Task will be canceled. Consider the following:

```
Task antecedent = Task.Factory.StartNew( () => {throw new Exception("boo!");});
bool ran = false;
Task c1 = antecedent.ContinueWith( _ => {ran = true;},
        TaskContinuationOptions.NotOnFaulted);
```

Since **antecedent** ends in a faulted state, and **c1** is created with the **NotOnFaulted** triggering option, **c1** is canceled rather than run. The final value of **ran** is **false**.

NotOnCanceled

This option instructs the continuation Task to **not** fire if the antecedent is canceled. If the antecedent is canceled, the continuation Task will be canceled. Consider the following:

```
Task antecedent = Task.Factory.StartNew( () => {throw new Exception("boo!");});
bool ran = false;
Task c1 = antecedent.ContinueWith( _ => {},
        TaskContinuationOptions.NotOnFaulted);
Task c2 = c1.ContinueWith( _ => {ran = true;},
        TaskContinuationOptions.NotOnCanceled);
```

Since **antecedent** ends in a faulted state, and **c1** is created with the **NotOnFaulted** triggering option, **c1** is canceled rather than run. Since **c2** is created with the **NotOnCanceled** triggering option, and its antecedent (**c1**) is canceled, then **c2** is canceled rather than run. The final value of **ran** is **false**.

OnlyOnRanToCompletion

This option instructs the continuation Task to fire **only** if the antecedent runs to completion. If the antecedent does not run to completion, the continuation Task will be canceled. Consider the following:

```
Task antecedent = Task.Factory.StartNew( () => {throw new Exception("Faulted!");});
bool ran = false;
Task c1 = antecedent.ContinueWith( _ => {ran = true;},
        TaskContinuationOptions.OnlyOnRanToCompletion);
```

Since **antecedent** ends in a Faulted state (and therefore does not run to completion), and **c1** is created with the **OnlyOnRanToCompletion** triggering option, **c1** is canceled rather than run. The final value of **ran** is **false**.

Useful Application: Only Run a Continuation if its Antecedent Produced Useful Data

This option is useful when you only want to run the continuation if the antecedent provided useful, complete data.

```
Task<int> t1 = Task<int>.Factory.StartNew( () =>
{
        … do some stuff …
        return someResult;
});

Task c1 = t1.ContinueWith(antecedent =>
{
        doSomethingWith(antecedent.Result);
}, TaskContinuationOptions.OnlyOnRanToCompletion);
```

OnlyOnFaulted

This option instructs the continuation Task to fire **only** if the antecedent is faulted (i.e., the antecedent's delegate throws an exception). If the antecedent does not end in a Faulted state, the continuation Task will be canceled. Consider the following:

```
Task antecedent = Task.Factory.StartNew( () => {});
bool ran = false;
Task c1 = antecedent.ContinueWith( _ => {ran = true;},
        TaskContinuationOptions.OnlyOnFaulted);
```

Since **antecedent** ends normally (and therefore does not end in a Faulted state), and **c1** is created with the **OnlyOnFaulted** triggering option, **c1** is canceled rather than run. The final value of **ran** is **false**.

Useful Application: Cleaning Up After Faulted Tasks

It is a common pattern to use this triggering option to run a "mop-up" continuation if the antecedent throws an exception. In this scenario, the antecedent is typically a "fire-and-forget" type of Task, and by

observing the antecedent's exception, the continuation logic prevents that exception from being thrown from the finalizer thread.

```
Task t1 = Task.Factory.StartNew( () => { … });

Task c1 = t1.ContinueWith(antecedent =>
{
        LogAndIgnore(antecedent.Exception);
},
TaskContinuationOptions.OnlyOnFaulted |
TaskContinuationOptions.ExecuteSynchronously);
```

OnlyOnCanceled

This option instructs the continuation Task to fire **only** if the antecedent is canceled. If the antecedent is not canceled, the continuation Task will be canceled. Consider the following:

```
Task antecedent = Task.Factory.StartNew( () => {});
bool ran = false;
Task c1 = antecedent.ContinueWith( _ => {ran = true;},
        TaskContinuationOptions.OnlyOnCanceled);
```

Since **antecedent** ends normally (and therefore is not canceled), and **c1** is created with the **OnlyOnCanceled** triggering option, **c1** is canceled rather than run. The final value of **ran** is **false**.

A Detailed Example

Consider the following example:

```
int count = 0;
Task A = Task.Factory.StartNew( () => {count++;});
Task B = A.ContinueWith( _ => {count++;},
        TaskContinuationOptions.OnlyOnFaulted);
Task C = B.ContinueWith( _ => {count++;},
        TaskContinuationOptions.NotOnCanceled);
Task D = C.ContinueWith( _ => {count++;},
        TaskContinuationOptions.NotOnRanToCompletion);
Task E = D.ContinueWith( _ => {count++;},
        TaskContinuationOptions.OnlyOnCanceled);
Task F = E.ContinueWith( _ => {count++;},
        TaskContinuationOptions.None);
```

What is the final value of count? Let's work through it:

- A completes. count => 1.
- B is canceled due to triggering mismatch with A (A completes, B is OnlyOnFaulted). count remains at 1.

- C is canceled due to triggering mismatch with B (B is Canceled, C is NotOnCanceled). count remains at 1.
- D runs. count => 2.
- E is canceled due to triggering mismatch with D (D completes, E is OnlyOnCanceled). count remains at 2.
- F runs. count => 3.

The final value of count = 3.

Anti-Pattern: Don't Combine Triggering Options

It seems self-evident that one should not combine "OnlyOn" options with each other when creating a continuation Task. "Only" implies a single triggering mode. Combining multiple "OnlyOn" options would result in an ArgumentOutOfRangeException being thrown at run time.

While it is permissible to combine the "NotOn" options with each other, it is bad form to do so. Here are the reasons why: (1) You can't combine all three "NotOn" options, as that would signify "don't fire under any condition", and would result in an ArgumentOutOfRangeException being thrown at run time, and (2) combining any two "NotOn" options is semantically equivalent to a single "OnlyOn" option.

For example, this combination of options would result in an ArgumentOutOfRangeException being thrown at run time:

```
Task ct = antecedent.ContinueWith( () => {},
        TaskContinuationOptions.NotOnRanToCompletion |
        TaskContinuationOptions.NotOnFaulted |
        TaskContinuationOptions.NotOnCanceled);
```

And this combination of "NotOn" options:

```
Task ct = antecedent.ContinueWith( () => {},
        TaskContinuationOptions.NotOnFaulted |
        TaskContinuationOptions.NotOnCanceled);
```

is equivalent to this usage of a single "OnlyOn" option:

```
Task ct = antecedent.ContinueWith( () => {},
        TaskContinuationOptions.OnlyOnRanToCompletion);
```

By similar logic, "OnlyOn" options should not be combined with "NotOn" options. At best, the "NotOn" option can be inferred from the "OnlyOn" option and is therefore superfluous. At worst, the "NotOn" option conflicts with the "OnlyOn" option, which will result in an ArgumentOutOfRangeException being thrown at run time.

ExecuteSynchronously

By default, a continuation Task will be started asynchronously with respect to its antecedent. That is, once the antecedent completes, the continuation Task will be submitted to the scheduler for execution. However, if the continuation Task is created with the ExecuteSynchronously option, it is run "inline" in a synchronous fashion. Such a synchronous continuation Task will run in one of two ways:

- If the antecedent is not complete when the continuation is created, the continuation will execute on the same thread that causes its antecedent to transition into a final state, immediately after the antecedent completes.
- If the antecedent is already complete when the continuation is created, the continuation will run immediately on the thread creating the continuation.

Such inline execution performs better than regular synchronous execution, as it skips the relatively heavyweight steps of submitting the continuation to the scheduler and de-queuing the continuation from the scheduler. As a rule, a synchronous continuation Task (i.e., a continuation Task created with the ExecuteSynchronously option) should be short and simple.

Useful Application: Optimize the Execution of Any Short-Running Continuation Task

Here is an example of how ExecuteSynchronously might be effectively employed:

```
Task<int> t1 = Task<int>.Factory.StartNew(…);
Task<int> t2 = Task<int>.Factory.StartNew(…);
Task<int> t3 = Task<int>.Factory.StartNew(…);
Task<int> adder = Task.Factory.ContinueWhenAll(new Task<int> [] {t1, t2, t3},
antecedents =>
{
        int sum = 0;
        for(int i=0; i<antecedents.Length; i++) sum += antecedents[i].Result;
        return sum;
}, TaskContinuationOptions.ExecuteSynchronously);
```

In the example above, Task **adder** is created to combine the results of Tasks **t1**, **t2** and **t3**. Since the continuation basically just adds three integers, it is a good candidate for running synchronously. (For simplicity's sake, we will assume that none of t1/t2/t3 throws an exception.)

Anti-Pattern: Don't Combine ExecuteSynchronously with LongRunning

As a rule, continuation Tasks that are executed synchronously should be "short and sweet". And it stands to reason that any continuation Task created with the LongRunning option would be, well, long-running. Therefore, an ArgumentOutOfRangeException will be thrown at run time if an attempt is made to create a continuation Task with both of these options.

Anti-Pattern: Don't Use ExecuteSynchronously for Long-Running Continuations

Again, the ExecuteSynchronously option should only be used for continuation Tasks with short, simple delegates. Continuations are occasionally run in finally{} blocks, which makes them uninterruptable

during AppDomain teardown. So a time-consuming synchronous continuation can potentially delay AppDomain teardown.

Anti-Pattern: Don't Create Long Chains of Synchronous Continuations

If you have a chain of N continuations, each of which is created with the ExecuteSynchronously option, it is the same as having N-deep call chain. (It's actually more like N*M-deep, where M is the number of internal function calls that accompanies each synchronous continuation execution). If N is sufficiently large, you will run out of room on the stack for function call frames, and stack overflow will result.

None

By default, a continuation Task:

- Is always started regardless of the final status of its antecedent(s)
- Runs asynchronously
- Is not created with any of the Task behavior options (LongRunning, PreferFairness or AttachedToParent)

Specifying TaskContinuationOptions.None at Task continuation creation merely preserves those defaults (so long as None is used in isolation, and not combined with any other TaskContinuationOptions).

Useful Application: Specifying Default Options When a TaskContinuationOptions Parameter is Required.

This option can come in handy when a TaskContinuationOptions parameter is required, but no variance from the default is desired. For example, if one wants to specify a TaskScheduler and a CancellationToken for a continuation, one must also provide a TaskContinuationOptions parameter:

```
Task c1 = someTask.ContinueWith( _ => {…},
        someCancellationToken,
        TaskContinuationOptions.None,
        someCustomTaskScheduler);
```

ParallelOptions

By default, the Parallel operations (Parallel.Invoke, Parallel.For and Parallel.ForEach) execute with the following characteristics:

- Any necessary support Tasks are spawned on the default TaskScheduler
- The degree of parallelism is unbounded
- There is no way to externally cancel the operation (though it can be canceled internally via Stop, Break or an exception being thrown)

The ParallelOptions type allows the caller to override the defaults for these operations.

TaskScheduler

By default, a Parallel operation runs all of its internally spawned Tasks on the default TaskScheduler. The ParallelOptions.TaskScheduler property can be used to specify an alternative TaskScheduler to a Parallel operation:

```
ParallelOptions options = new ParallelOptions { TaskScheduler = someTaskScheduler };
Parallel.Invoke(options,
        () => { action1(); },
        () => { action2(); },
        () => { action3(); },
        () => { action4(); });
```

In this example, any Tasks launched to service the component actions will be queued to someTaskScheduler, rather than the default scheduler.

Note that it is legal to specify a value of null for ParallelOptions.TaskScheduler. The meaning of this is "use the ambient TaskScheduler". Consider this example:

```
Task.Factory.StartNew( () =>
{
    […]
    ParallelOptions options = new ParallelOptions {TaskScheduler = null };
    Parallel.Invoke(options,
        () => { […] },
        () => { […] },
        …
        () => { […] });
}, CancellationToken.None, TaskCreationOptions.None, someCustomTaskScheduler);
```

In the code above, the Parallel.Invoke is instructed, via ParallelOptions.TaskScheduler being set to null, to use the ambient (or "current") TaskScheduler. The current TaskScheduler in this case would be someCustomTaskScheduler, because the enclosing Task was run on someCustomTaskScheduler.

Useful Application: Queuing Worker Tasks to a Custom TaskScheduler

Obviously, the most common usage of this option would be to schedule internally generated Tasks onto some TaskScheduler other than the default TaskScheduler. The specific reasons for doing this are as varied as the number of customized TaskSchedulers that one can imagine. A couple of possibilities are:

- routing internally generated Tasks to a throttling TaskScheduler that limits overall concurrency
- routing internally generated Tasks to a TaskScheduler that spawns a thread-per-Task[3]

[3] Though if you did this, you would also want to specify MaxDegreeOfParallelism to cap the thread count at something reasonable.

Anti-Pattern: Never Use the UI Scheduler as the TaskScheduler for Parallel Operations

This, for example, would be bad form for a GUI application:

```
ParallelOptions options = new ParallelOptions();
options.TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()};
Parallel.For(0, 1000000, options, i =>
{
        doSomething(i);
});
```

The reasons that this is a bad idea are:

(1) All of the Tasks spawned from Parallel.For would run on the main UI thread, sequentially. Thus, your Parallel.For would effectively run sequentially, which surely would not have been the intent.
(2) All of that work running on the main UI thread would render your GUI unresponsive.

MaxDegreeOfParallelism

By default, a Parallel operation will use all available parallelism to execute. The ParallelOptions.MaxDegreeOfParallelism property allows a user to override that behavior:

```
ParallelOptions options = new ParallelOptions { MaxDegreeOfParallelism = 2 };
Parallel.For(0, 1000, options, delegate(int i)
{
    someOperation(i);
});
```

In the example above, only two Tasks (at **most**) will be concurrently utilized to process the Parallel.For loop.

Here a couple of other interesting things to note about the MaxDegreeOfParallelism property:

- There is no guarantee that a Parallel.For/ForEach/Invoke operation will utilize more than a single Task concurrently. On a busy system, that may be all that you get.
- The MaxDegreeOfParallelism property only applies to a single Parallel operation. For example, in the code below, the inner Parallel.For operation has no limits on its degree of parallelism, even though the outer parallel.For is constrained to running on two Tasks concurrently.

```
ParallelOptions options = new ParallelOptions { MaxDegreeOfParallelism = 2 };
Parallel.For(0, 1000, options, delegate(int i)
{
    Parallel.For(0, 1000, delegate(int j)
    {
        doSomeOperation(i, j);
    });
});
```

Useful Application: Capping Thread Injection

The MaxDegreeOfParallelism property of ParallelOptions can be used to cap ThreadPool thread injection when using the default TaskScheduler. Consider the following code:

```
Parallel.For(0, 10000, i =>
{
        longComputeBoundOperation(i); // maybe 10 secs per iteration
});
```

The code above will result in the ThreadPool slowly injecting worker threads as it sees that no work is completing. This will result in multiple instances of **longComputeBoundOperation()** running on the same core, competing for CPU and other resources; the end result is general performance degradation. If you want to cap the total number of compute threads associated with the Parallel.For to, say, the number of cores, you could do the following:

```
int numProcs = Environment.ProcessorCount;
ParallelOptions options = new ParallelOptions {MaxDegreeOfParallelism = numProcs;};
Parallel.For(0, 10000, options, i =>
{
        longComputeBoundOperation(i); // maybe 10 secs per iteration
});
```

CancellationToken

By default, a Parallel operation cannot be canceled. True, it can exit with an exception, or as a result of a Stop or Break call (in the case of Parallel.For/ForEach), but by default it cannot be externally canceled. The ParallelOptions.CancellationToken property allows the user to provide a CancellationToken to a Parallel operation, the cancellation of which (through its owning CancellationTokenSource) will cancel the operation and throw an OperationCanceledException.

Useful Application: Cancellation Support in GUIs

One common use of this feature is to support "Cancel" buttons in graphical user interfaces (GUIs). For example, a GUI might have buttons for "Start" and "Cancel". The code associated with pressing the "Start" button might look like this:

```
void StartButton_Click(…)
{
        [ cts is a CancellationTokenSource declared in some central location ]
        cts = new CancellationTokenSource();
        TaskScheduler ui_scheduler =
                TaskScheduler.FromCurrentSynchronizationContext();
```

```
ParallelOptions options = new ParallelOptions { CancellationToken = cts.Token };
int count = 0;
int length = someArray.Length;
// Launch a Task in the background to perform a Parallel.For
Task work = Task.Factory.StartNew( () =>
{
        Parallel.For(0, length, options, index =>
        {
                process(someArray[index]);
                int newCount = Interlocked.Increment(ref count);
                if( (newCount % 10) == 0) // update progressBar every 10
                {
                        Task.Factory.StartNew( () =>
                        {
                                progressBar.Value =
                                  (double) newCount * 100.0 / (double) length;
                        }, ui_scheduler);
                }
        });
}, cts.Token, TaskCreationOptions.None, TaskScheduler.Default);

// When "work" completes, hide the progressBar.
work.ContinueWith( antecedent =>
{
        progressBar.Visibility = Visibility.Hidden;
},
TaskContinuationOptions.ExecuteSynchronously);

}
```

and the code associated with pressing the "Cancel" button might look like this:

```
void CancelButton_Click(…)
{
        cts.Cancel();
}
```

If the "Cancel" button is pressed while the Parallel.ForEach loop is still executing, the Parallel.For loop will be discontinued.

References

[1] Eric Eilebrect, "CLR ThreadPool Improvements: Part 1",
http://blogs.msdn.com/ericeil/archive/2009/04/23/clr-4-0-threadpool-improvements-part-1.aspx

[2] Joe Duffy, “Building a custom thread pool (series, part 2): a work stealing queue”, http://www.bluebytesoftware.com/blog/2008/08/12/BuildingACustomThreadPoolSeriesPart2AWorkStealingQueue.aspx

[3] Daniel Moth, “New and Improved CLR 4 Thread Pool Engine”, http://www.danielmoth.com/Blog/2008/11/new-and-improved-clr-4-thread-pool.html

[4] Stephen Toub, “TaskCreationOptions.PreferFairness”, http://blogs.msdn.com/pfxteam/archive/2009/07/07/9822857.aspx

This material is provided for informational purposes only. Microsoft makes no warranties, express or implied. ©2009 Microsoft Corporation.